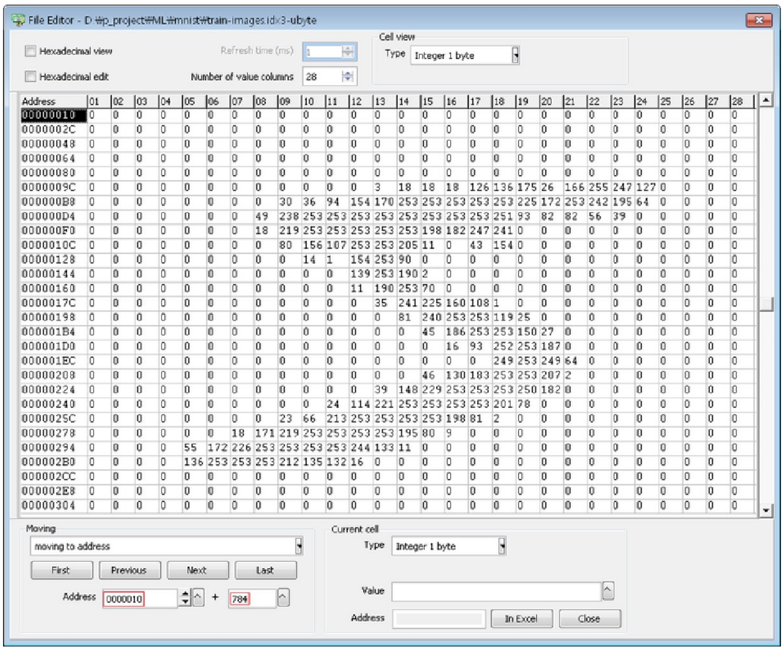
Option Pricing using Neural Network in *Mathematica*

김세기 (성균관대학교)

Digit Classification(Neural Network)

Use the MNIST database of handwritten digits to train a **convolutional network** to predict the digit given an image. First obtain the training and validation data.

육필(손글씨) 숫자 MNIST 데이터베이스를 사용하여 이미지에서 주어진 숫자를 예측하는 **합성곱 네트워크**를 훈련합니다.
먼저, 학습 데이터와 검증 데이터를 검색합니다



File Editor - D:₩p_project₩ML₩mnist₩train-images.idx3-ubyte

□ Hexadecimal view   Refresh time (ms) 1

□ Hexadecimal edit   Number of value columns 28

Cell view Type: Integer 1 byte

Address	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
00000010	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0000002C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
00000048	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
00000064	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
00000080	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0000009C	0	0	0	0	0	0	0	0	0	0	0	0	3	18	18	18	126	136	175	26	166	255	247	127	0	0	0	0
000000B8	0	0	0	0	0	0	0	0	30	36	94	154	170	253	253	253	253	253	225	172	253	242	195	64	0	0	0	0
000000D4	0	0	0	0	0	0	0	49	238	253	253	253	253	253	253	253	253	251	93	82	82	56	39	0	0	0	0	0
000000F0	0	0	0	0	0	0	0	18	219	253	253	253	253	253	198	182	247	241	0	0	0	0	0	0	0	0	0	0
0000010C	0	0	0	0	0	0	0	0	80	156	107	253	253	205	11	0	43	154	0	0	0	0	0	0	0	0	0	0
00000128	0	0	0	0	0	0	0	0	0	14	1	154	253	90	0	0	0	0	0	0	0	0	0	0	0	0	0	0
00000144	0	0	0	0	0	0	0	0	0	0	0	139	253	190	2	0	0	0	0	0	0	0	0	0	0	0	0	0
00000160	0	0	0	0	0	0	0	0	0	0	0	11	190	253	70	0	0	0	0	0	0	0	0	0	0	0	0	0
0000017C	0	0	0	0	0	0	0	0	0	0	0	0	35	241	225	160	108	1	0	0	0	0	0	0	0	0	0	0
00000198	0	0	0	0	0	0	0	0	0	0	0	0	0	81	240	253	253	119	25	0	0	0	0	0	0	0	0	0
000001B4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	45	186	253	253	150	27	0	0	0	0	0	0	0	0
000001D0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	93	252	253	187	0	0	0	0	0	0	0	0
000001EC	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	249	253	249	64	0	0	0	0	0	0	0
00000208	0	0	0	0	0	0	0	0	0	0	0	0	0	0	46	130	183	253	253	207	2	0	0	0	0	0	0	0
00000224	0	0	0	0	0	0	0	0	0	0	0	0	39	148	229	253	253	253	250	182	0	0	0	0	0	0	0	0
00000240	0	0	0	0	0	0	0	0	0	0	24	114	221	253	253	253	253	201	78	0	0	0	0	0	0	0	0	0
0000025C	0	0	0	0	0	0	0	0	23	66	213	253	253	253	253	198	81	2	0	0	0	0	0	0	0	0	0	0
00000278	0	0	0	0	0	0	18	171	219	253	253	253	253	195	80	9	0	0	0	0	0	0	0	0	0	0	0	0
00000294	0	0	0	0	55	172	226	253	253	253	253	244	133	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0
000002B0	0	0	0	0	136	253	253	253	212	135	132	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
000002CC	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
000002E8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
00000304	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

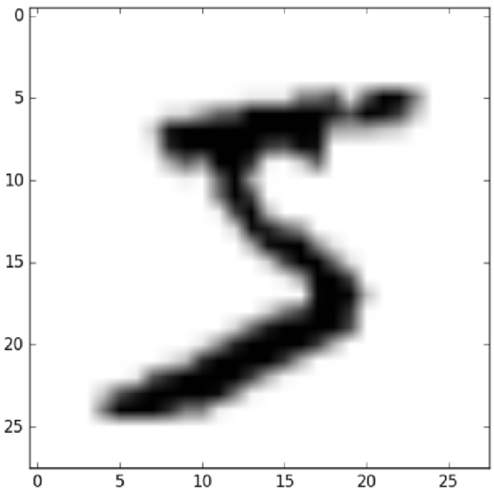
Moving: moving to address

Address 0000010 + 784

Current cell Type: Integer 1 byte

Value

Address

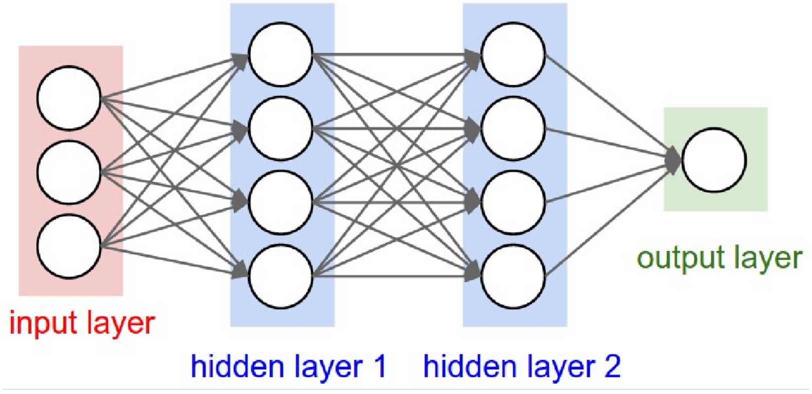


In[ ]:=
```
resource = ResourceObject["MNIST"];
trainingData = ResourceData[resource, "TrainingData"];
testData = ResourceData[resource, "TestData"];
```

****

Fully Connected Neural Network (FCNN)

****



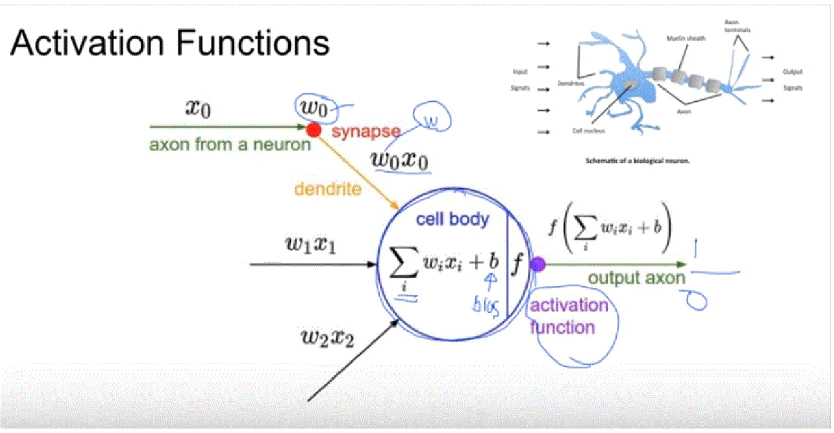
The above neural network is called a **3 - layer neural network** with three inputs, two hidden layers of 4 neurons each and one output layer.

Notice that there are connections (synapses) between neurons across layers, but not within a layer.

Linear Layer :

w·x + b where **w** is a weight and **b** is a bias with input **x**.

In[∘]:=



Activation Functions

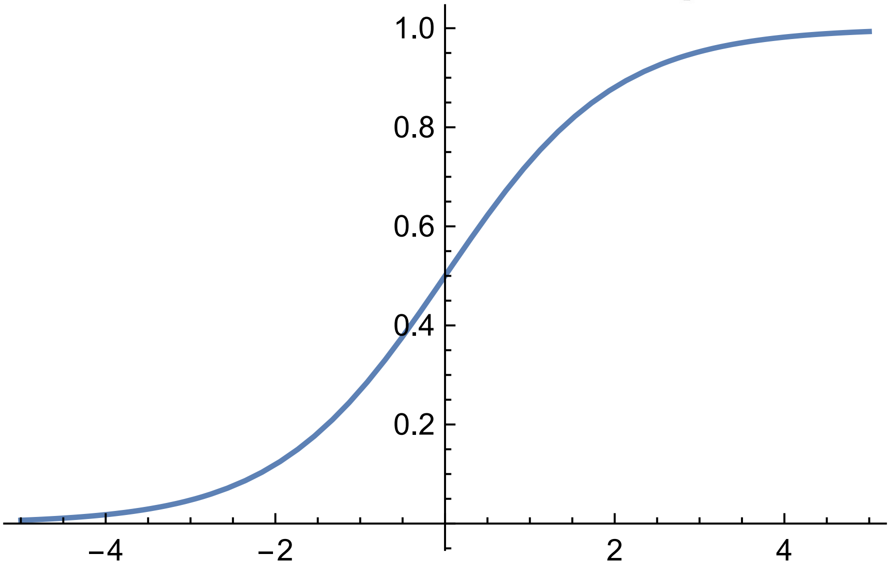
Activation function :

In computational networks, the activation function of a node defines the output of that node given an input or set of inputs.
A standard computer chip circuit can be seen as a digital network of activation functions that can be "ON" (1) or "OFF" (0), depending on input. This is similar to the behavior of the linear perceptron in neural networks. However, only nonlinear activation functions allow such networks to compute nontrivial problems using only a small number of nodes. In artificial neural

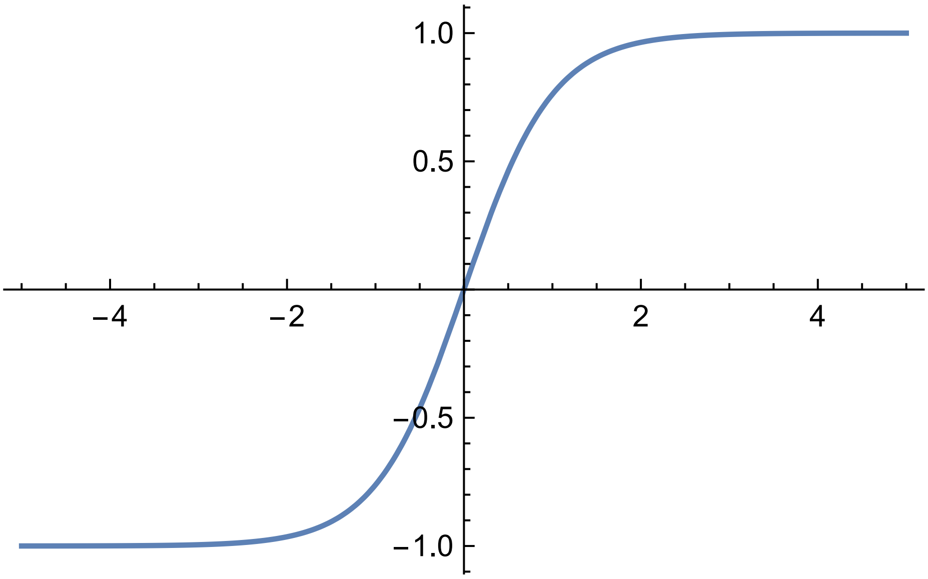
$$f(x) = \frac{1}{1+e^{-x}}$$

networks this function is also called the transfer function.

e.g. 1. Sigmoid(Logistic) $f(x) = \dfrac{1}{1+e^{-x}}$

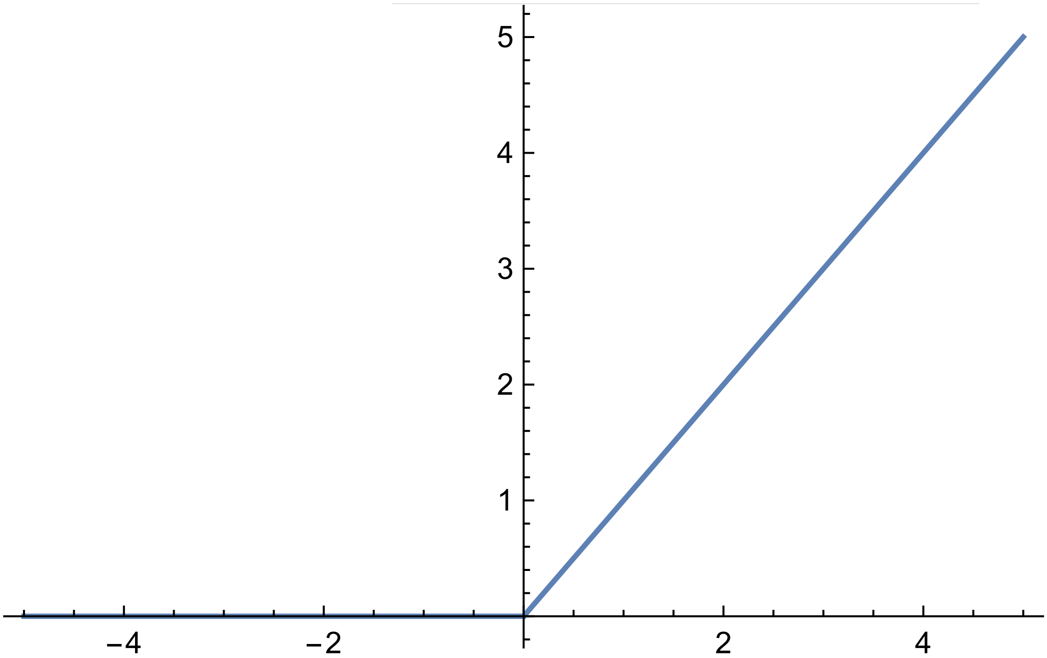


2. Tanh $f(x) = \tanh(x) = \dfrac{2}{1+e^{-2x}} - 1$



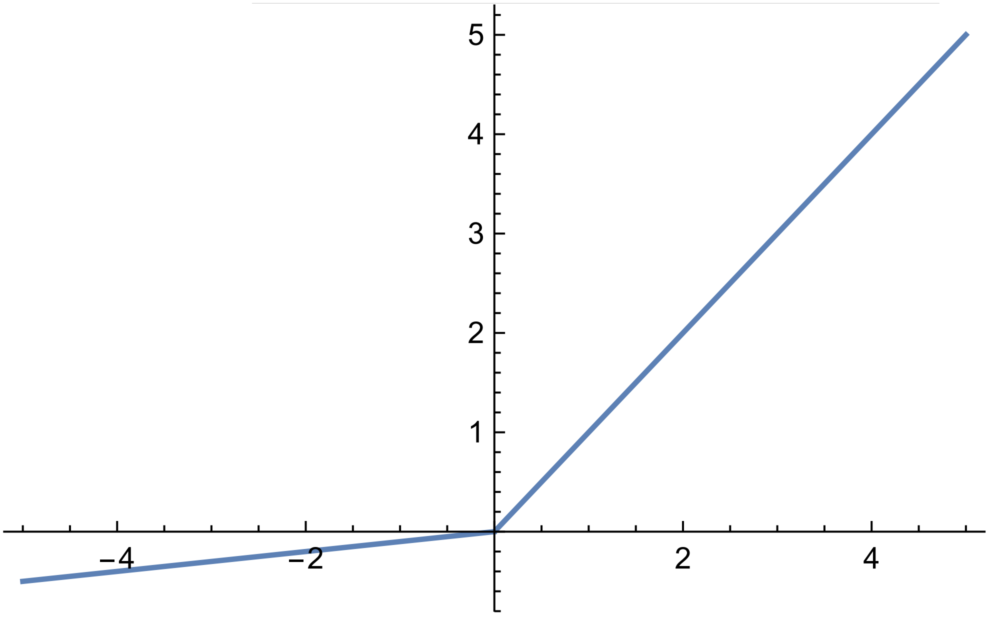
3. ReLU (Ramp) $f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$

$$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$$

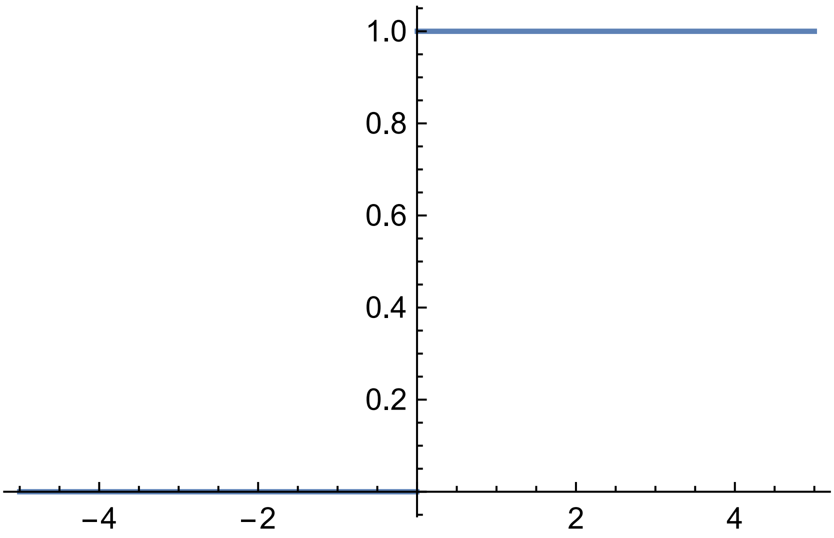


4. Leaky ReLU (Leaky rectified linear unit)

$$f(x) = \begin{cases} 0.01x & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$$



5. Binary Step function (Heaviside/unit step function)



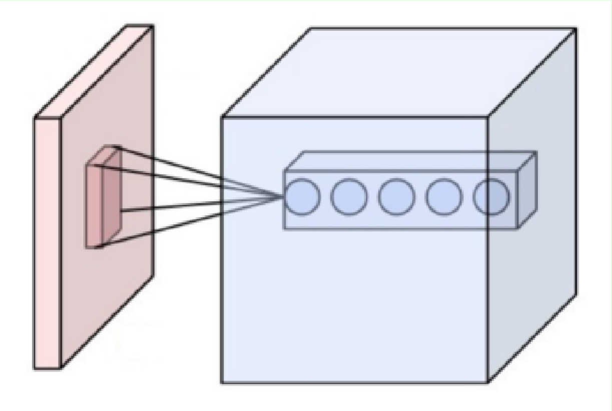
Ramp[x] gives x if $x \geqslant 0$
and 0 otherwise.
(= > ReLU : Rectified Linear Unit)

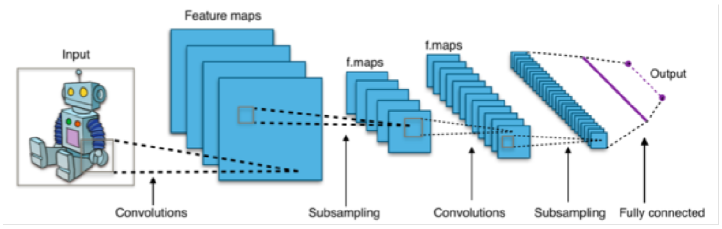
****

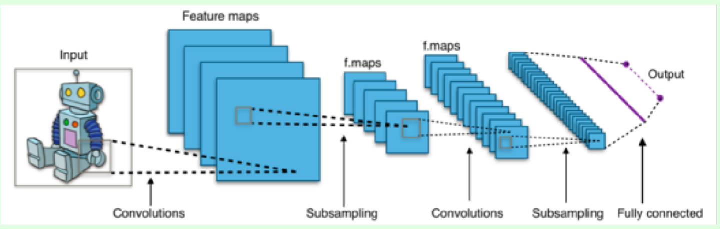
Convolutional Neural Network (CNN)

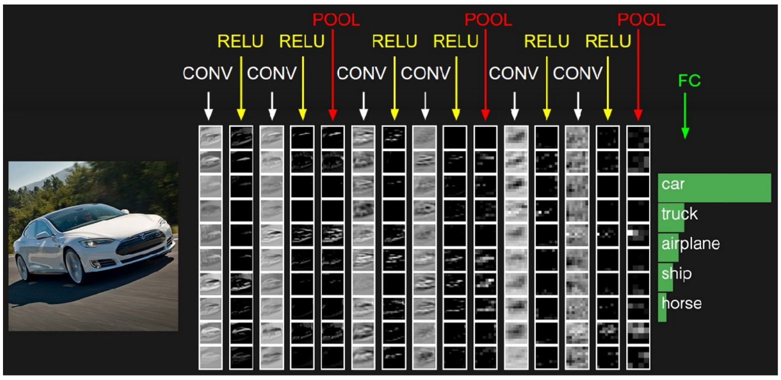
****

Define a **convolutional neural network** that takes
in 28×28 grayscale images as input.
입력으로 28*28 그레이 스케일 이미지를 가지는
합성곱 분류 신경망을 정의합니다.









```
In[∘]:= lenet =
         NetChain[{ConvolutionLayer[20, 5], Ramp, PoolingLayer[2, 2], ConvolutionLayer[50, 5],
           Ramp, PoolingLayer[2, 2], FlattenLayer[], 500, Ramp, 10, SoftmaxLayer[]},
          "Output" → NetDecoder[{"Class", Range[0, 9]}],
          "Input" → NetEncoder[{"Image", {28, 28}, "Grayscale"}]]

Out[∘]=
        NetChain[ ⊞ uninitialized   Input port:    image
                                    Output port:   class ]
```

Avoid Overfitting Using a Hold-Out Set

홀드 아웃 셋을 이용한 과잉 적합 회피

Use the ValidationSet option to NetTrain to ensure that the trained net does not overfit the input data.

This is commonly referred to as **a test or hold-out dataset.**

NetTrain의 ValidationSet 옵션을 이용하여 훈련된 네트워크가 입력 데이터에 과잉 적합하지 않도록 합니다.

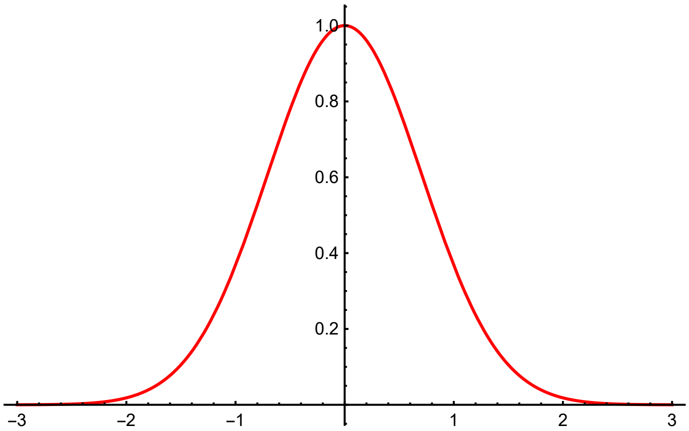
이는 일반적으로 테스트 또는 홀드 아웃이라 지칭됩니다.

Create synthetic training data based on a Gaussian curve.

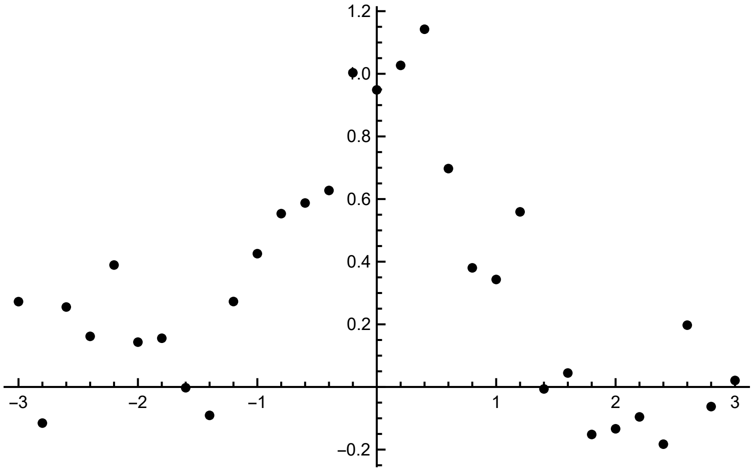
가우스 곡선을 기반으로 합성 훈련 데이터를 생성합니다.

NormalDistribution[μ, σ] represnts a normal(Gaussian) distribution with mean μ and standard deviation σ.

```
In[∘]:= gplot = Plot[Exp[-x^2], {x, -3, 3}, PlotStyle → Red]
Out[∘]=
```



```
In[∘]:= data = Table[x → Exp[-x^2] + RandomVariate[NormalDistribution[0, .15]], {x, -3, 3, .2}];
                       (* NormalDistribution [μ,σ] *)
        plot1 = ListPlot[List @@@ data, PlotStyle → Black]
Out[∘]=
```



```
In[∘]:= data
Out[∘]=
        {-3. → 0.272477, -2.8 → -0.115376, -2.6 → 0.25527, -2.4 → 0.161611,
         -2.2 → 0.389377, -2. → 0.143151, -1.8 → 0.155571, -1.6 → -0.00247843,
         -1.4 → -0.0906944, -1.2 → 0.272815, -1. → 0.42564, -0.8 → 0.553465,
         -0.6 → 0.587414, -0.4 → 0.627466, -0.2 → 1.00326, 0. → 0.948622, 0.2 → 1.02689,
         0.4 → 1.14227, 0.6 → 0.697314, 0.8 → 0.380323, 1. → 0.343238, 1.2 → 0.559159,
         1.4 → -0.00635132, 1.6 → 0.0445425, 1.8 → -0.151739, 2. → -0.133583,
         2.2 → -0.0956092, 2.4 → -0.182722, 2.6 → 0.197388, 2.8 → -0.0626517, 3. → 0.0208207}
```

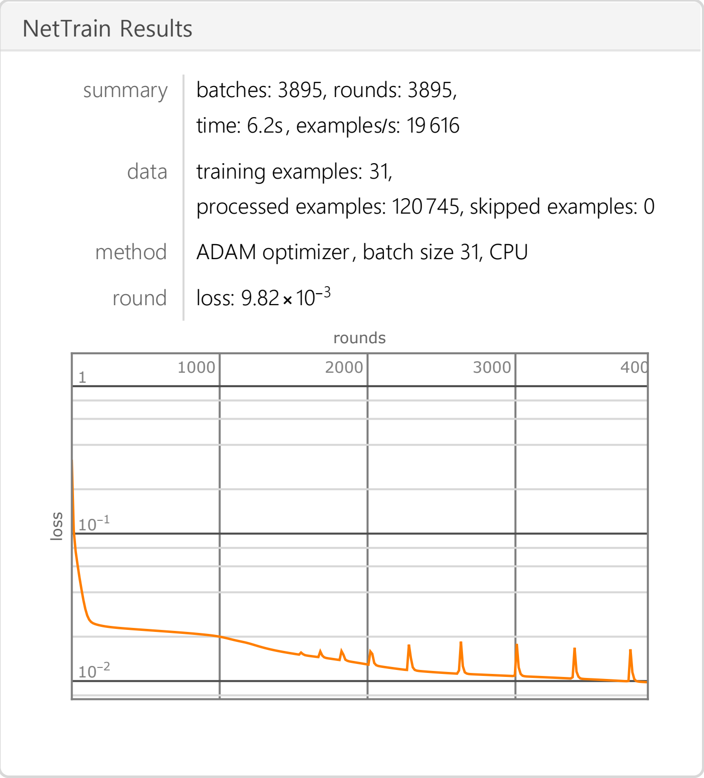
```
In[ ]:= List @@@ data
Out[ ]=
        {{-3., 0.272477}, {-2.8, -0.115376}, {-2.6, 0.25527},
         {-2.4, 0.161611}, {-2.2, 0.389377}, {-2., 0.143151}, {-1.8, 0.155571},
         {-1.6, -0.00247843}, {-1.4, -0.0906944}, {-1.2, 0.272815}, {-1., 0.42564},
         {-0.8, 0.553465}, {-0.6, 0.587414}, {-0.4, 0.627466}, {-0.2, 1.00326},
         {0., 0.948622}, {0.2, 1.02689}, {0.4, 1.14227}, {0.6, 0.697314},
         {0.8, 0.380323}, {1., 0.343238}, {1.2, 0.559159}, {1.4, -0.00635132},
         {1.6, 0.0445425}, {1.8, -0.151739}, {2., -0.133583}, {2.2, -0.0956092},
         {2.4, -0.182722}, {2.6, 0.197388}, {2.8, -0.0626517}, {3., 0.0208207}}

In[ ]:= Length[data]
Out[ ]=
        31
```

Train a net with a large number of parameters relative to the amount of training data.

훈련 데이터의 양에 비해 다수의 파라미터를 가지는 네트워크를 훈련합니다

```
In[ ]:= net = NetChain[{150, Tanh, 150, Tanh, 1}, "Input" → "Scalar", "Output" → "Scalar"];
        results1 = NetTrain[net, data, All, Method → "ADAM"]
Out[ ]=
```

NetTrainResultsObject[



NetTrain Results

summary: batches: 3895, rounds: 3895, time: 6.2s, examples/s: 19 616

data: training examples: 31, processed examples: 120 745, skipped examples: 0

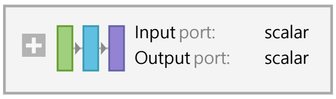
method: ADAM optimizer, batch size 31, CPU

round: loss: 9.82×10^{-3}

]

```
In[∘]:= net1 = results1["TrainedNet"]
Out[∘]=
```

NetChain[



Input port: scalar
Output port: scalar

]

여기서 150 은 neuron의 수.

```
In[∘]:= net1[-100.271]
        net1[100.398]
Out[∘]=
        0.255547
Out[∘]=
        -0.274822
```

NetChain[{$layer_1$, $layer_2$, …}]

specifies a neural net in which the output of $layer_i$ is connected to the input of $layer_{i+1}$.

"ADAM" stochastic gradient descent using
an adaptive learning rate that is invariant to
diagonal rescaling of the gradients

"RMSProp" stochastic gradient descent using
an adaptive learning rate derived from
exponentially smoothed average of gradient magnitude

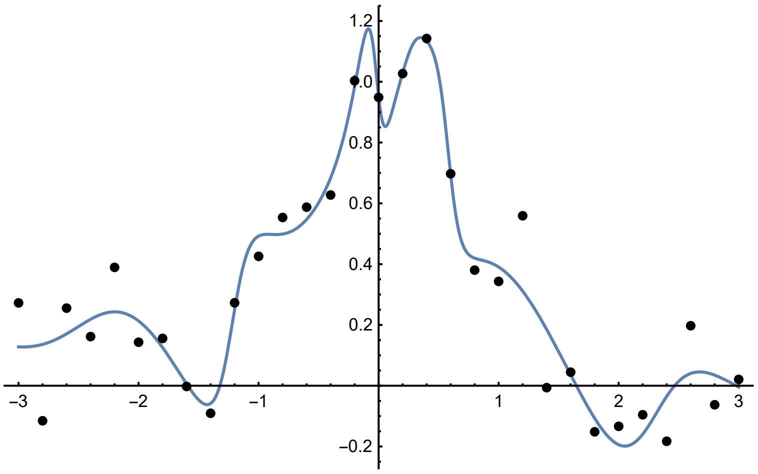
"SGD" ordinary stochastic gradient descent with momentum

"SignSGD" stochastic gradient descent for which
the magnitude of the gradient is discarded

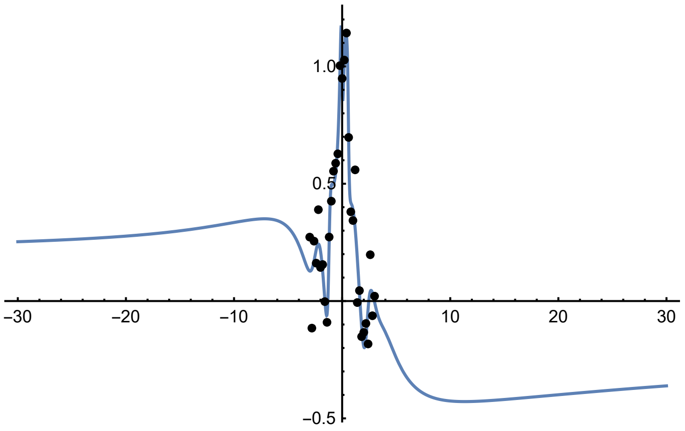
The resulting net overfits the data, learning the noise
in addition to the underlying function.

기반 함수 및 노이즈도 학습하는 결과 네트워크는 데이터에 과잉 적합합니다.

```
In[∘]:= Show[Plot[net1[x], {x, -3, 3}], plot1]
Out[∘]=
```



```
In[∘]:= Show[Plot[net1[x], {x, -30, 30}], plot1]
Out[∘]=
```



Subdivide the data into a training set and a hold-out validation set.
훈련 집합과 홀드 아웃 검증 집합으로 데이터를 분할합니다.

```
In[∘]:= data2 = RandomSample[data];
        {train2, test2} = TakeDrop[data2, 24];
```

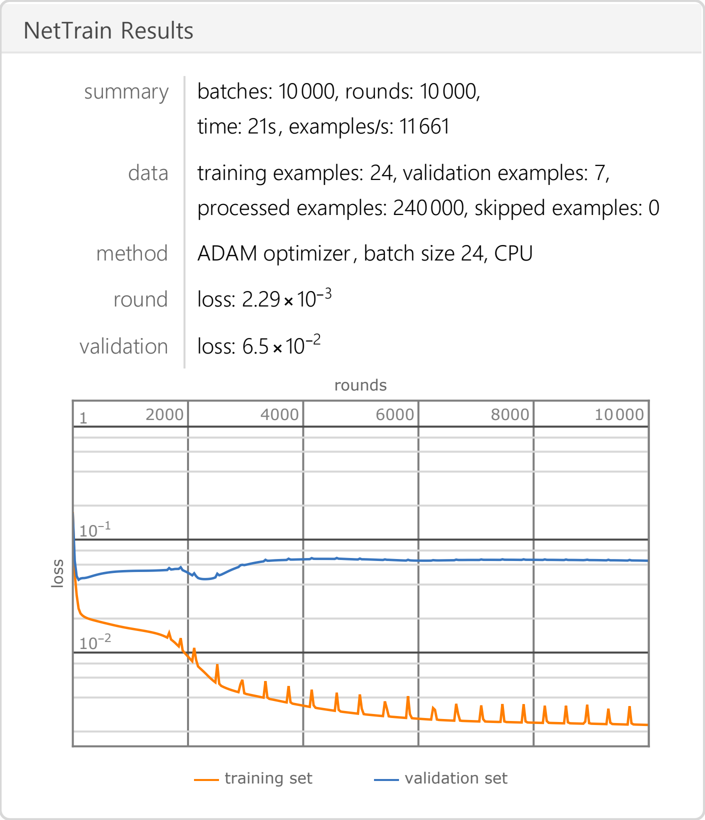
Use the ValidationSet option to have NetTrain select the net
that achieved the lowest validation loss during training.

훈련하는 동안 NetTrain가 최저 검증 손실을 달성한 네트워크를 선택하도록
ValidationSet 옵션을 사용합니다.

```
ValidationSet : the set of data on which to evaluate the model
                during training.
```

```
In[∘]:= results12 = NetTrain[net, train2, All, ValidationSet → test2]
Out[∘]=
```

NetTrainResultsObject[



NetTrain Results

summary: batches: 10 000, rounds: 10 000, time: 21s, examples/s: 11 661

data: training examples: 24, validation examples: 7, processed examples: 240 000, skipped examples: 0

method: ADAM optimizer, batch size 24, CPU

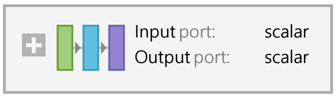
round: loss: 2.29×10^{-3}

validation: loss: 6.5×10^{-2}

]

```
In[∘]:= net2 = results12["TrainedNet"]
Out[∘]=
```

NetChain[



Input port: scalar
Output port: scalar

]

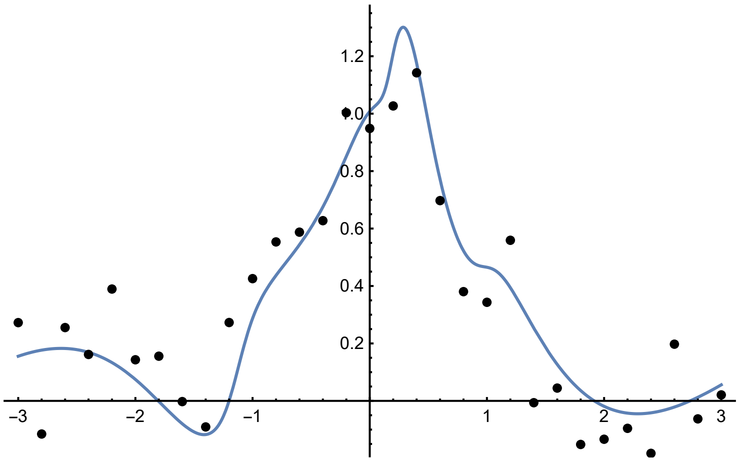
The result returned by NetTrain was the net that generalized best to points in the validation set, as measured by validation loss.

This penalizes overfitting, as the noise present in the training data is uncorrelated with the noise present in the validation set.

NetTrain에 의해 반환된 결과는 검증 손실에 의해 측정된 바와 같이, 검증 집합의 점들에 가장 잘 일반화된 네트워크입니다.

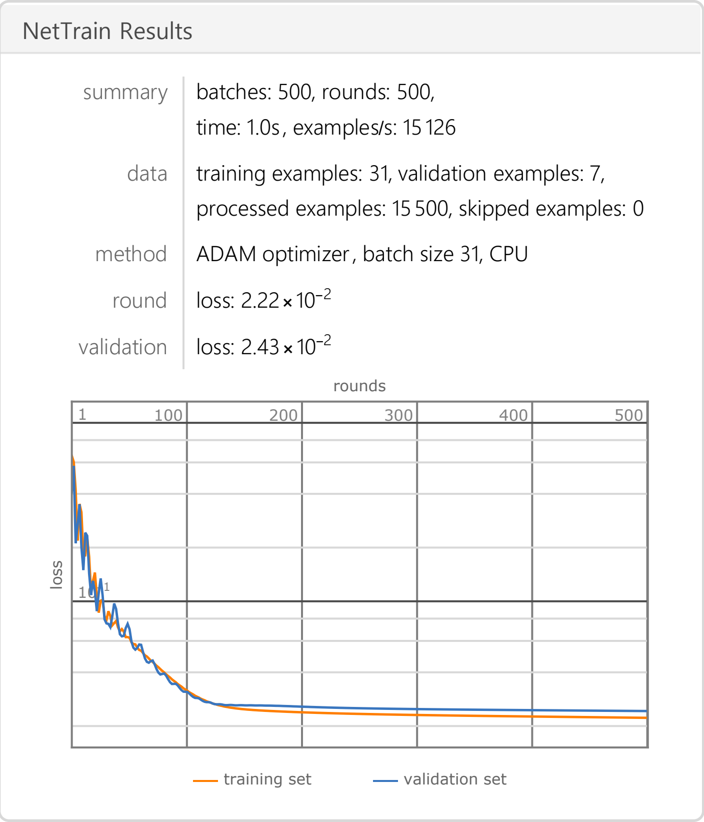
훈련 데이터에 존재하는 노이즈는 검증 집합에 존재하는 잡음과 전혀 상관 관계가 없음을 입증하는 것으로, 이는 과잉 적합에 벌칙을 주게 됩니다.

```
In[∘]:= Show[Plot[net2[x], {x, -3, 3}], plot1]
Out[∘]=
```



```
In[∘]:= testX = Keys[data];
        testY = Values[data];
        meanTestLoss[net_] := SquaredEuclideanDistance[net[testX], testY] / Length[testX];

In[∘]:= meanTestLoss[net1]
        meanTestLoss[net2]
Out[∘]=
        0.00981785
Out[∘]=
        0.0208565

In[∘]:= results2 = NetTrain[net, data, All, ValidationSet → test2, MaxTrainingRounds → 500]
Out[∘]=
```

NetTrainResultsObject[



NetTrain Results

summary batches: 500, rounds: 500,
time: 1.0s, examples/s: 15 126

data training examples: 31, validation examples: 7,
processed examples: 15 500, skipped examples: 0

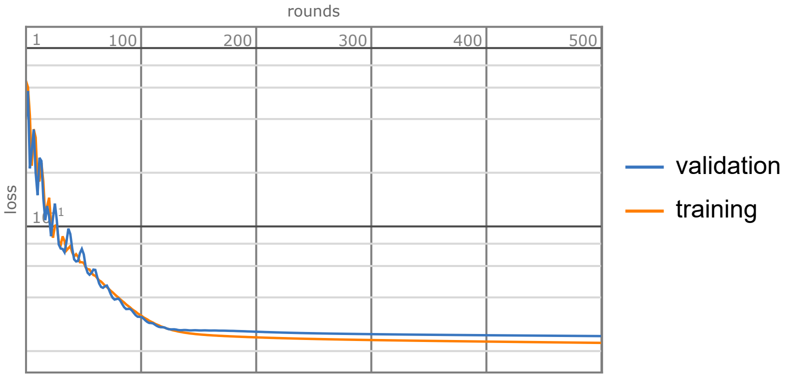
method ADAM optimizer, batch size 31, CPU

round loss: 2.22 × 10^{-2}

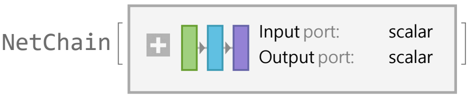
validation loss: 2.43 × 10^{-2}

]

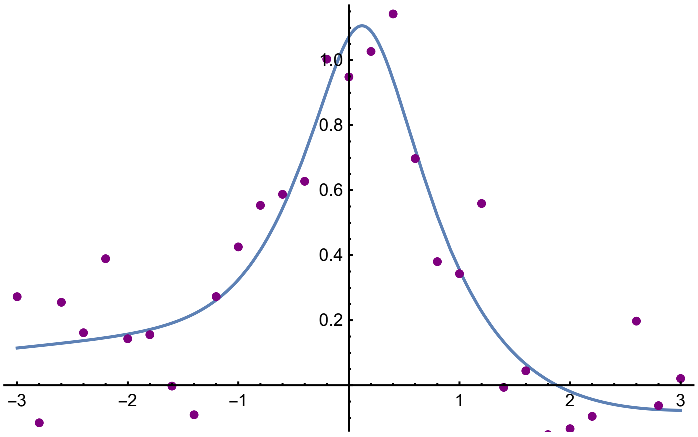
```
In[∘]:= results2["LossEvolutionPlot"]
Out[∘]=
```



```
In[∘]:= earlyStoppingNet = results2["TrainedNet"]
Out[∘]=
```


NetChain
Input port: scalar
Output port: scalar

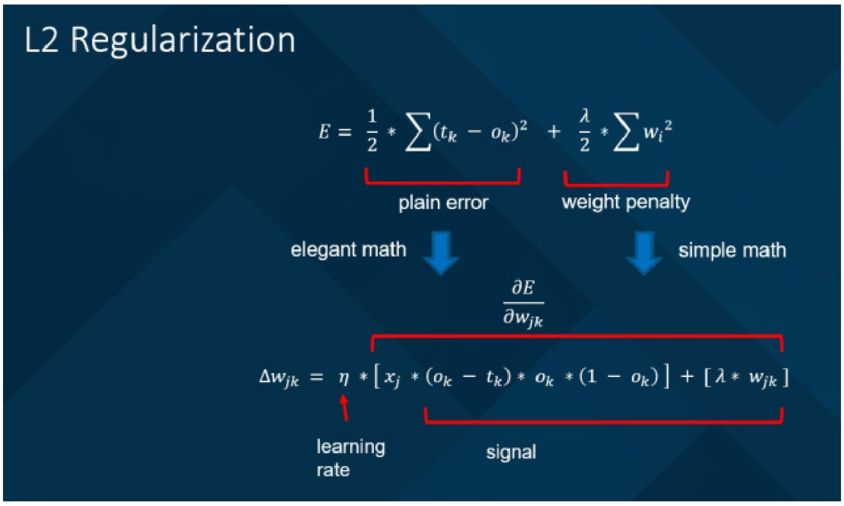
```
In[∘]:= Show[Plot[earlyStoppingNet[x], {x, -3, 3}], ListPlot[List @@@ data, PlotStyle → Purple]]
Out[∘]=
```



Another way to tackle overfitting is to use **L2 regularization**, which implicitly associates a loss with nonzero parameters in the net during training. This can be specified with a Method option to NetTrain.

과잉 적합을 다루는 또다른 방법으로, 훈련 도중 네트워크에 non-zero 매개 변수를 사용하여 손실을 암시적으로 연관시키는 L2 정규화를 사용하는 것입니다. 이는 NetTrain의 옵션 Method를 사용하여 지정할 수 있습니다.

```
"L2Regularization"  the global loss associated with the L2 norm
                    of all learned tensors.
                        (l^2 - norm)
                    or  the global loss associated with
                            the L2 norm of all learned arrays
```



L2 Regularization

$$E = \underbrace{\frac{1}{2} * \sum (t_k - o_k)^2}_{\text{plain error}} + \underbrace{\frac{\lambda}{2} * \sum w_i^2}_{\text{weight penalty}}$$

elegant math

simple math

$$\Delta w_{jk} = \eta * \overbrace{[x_j * (o_k - t_k) * o_k * (1 - o_k)] + [\lambda * w_{jk}]}^{\frac{\partial E}{\partial w_{jk}}}$$

learning rate

signal

where $(t_k - \sigma(z))^2$

with $z = w * x + b$ and $\sigma(z) = \frac{1}{1 + e^{-z}}$

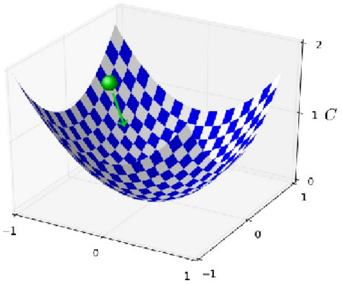
Note that $\sigma'(z) = \sigma(z)(1 - \sigma(z))$

where $\sigma'(z) = \frac{\partial \sigma}{\partial z}$.

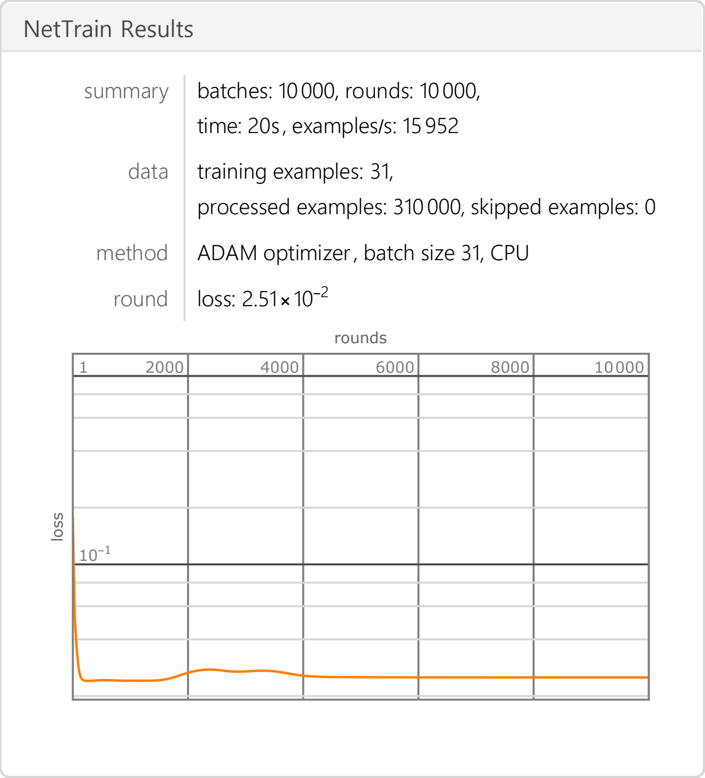
$w_{n+1} = w_n + \Delta w$ makes E less in value

where $\Delta w = \text{learning rate} * \frac{\partial E}{\partial w}$

where the learning rate is the size of steps to take in the direction of the derivative.



```
In[ ]:= results3 = NetTrain[net, data, All, Method → {"ADAM", "L2Regularization" → 0.01}]
Out[ ]=
NetTrainResultsObject[ ]
```



NetTrain Results

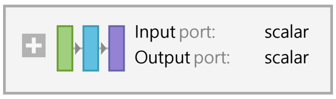
summary batches: 10 000, rounds: 10 000,
time: 20s, examples/s: 15 952

data training examples: 31,
processed examples: 310 000, skipped examples: 0

method ADAM optimizer, batch size 31, CPU

round loss: 2.51×10^{-2}

```
In[ ]:= net3 = results3["TrainedNet"]
Out[ ]=
NetChain[ ]
```



Input port: scalar
Output port: scalar

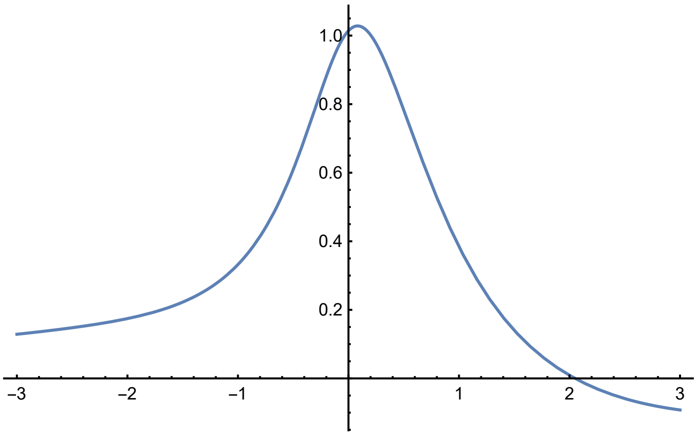
The suboption of "L2Regularization"
can be given in the following form :
"L2Regularization" → r
: use the value r for all weights (multiplier) in the net.

L2 regularization penalizes “complex” nets, as measured by the magnitudes of their parameters, which tends to reduce overfitting.

L2 정규화는 과잉적합을 감소시키는 경향이 있는 매개 변수의 크기에 의해 측정되는 “복소” 네트에 벌칙을 주는 것임.

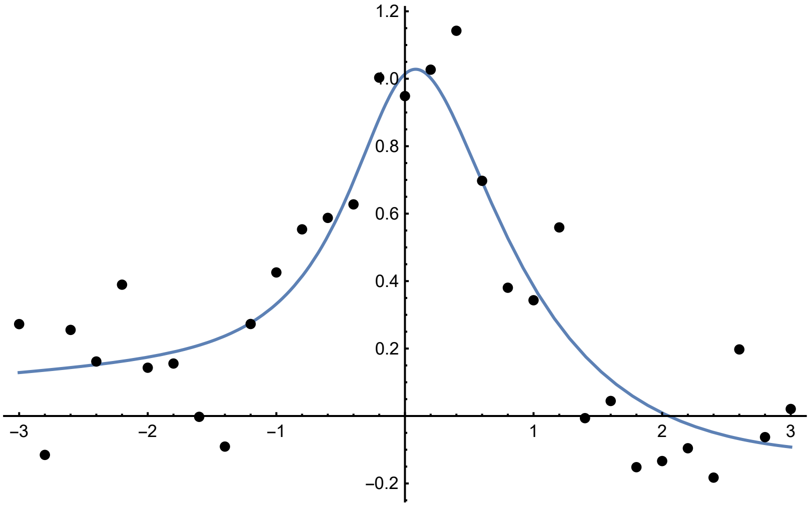
In[ ]:=
```
Plot[net3[x], {x, -3, 3}, PlotRange → All]
```

Out[ ]=



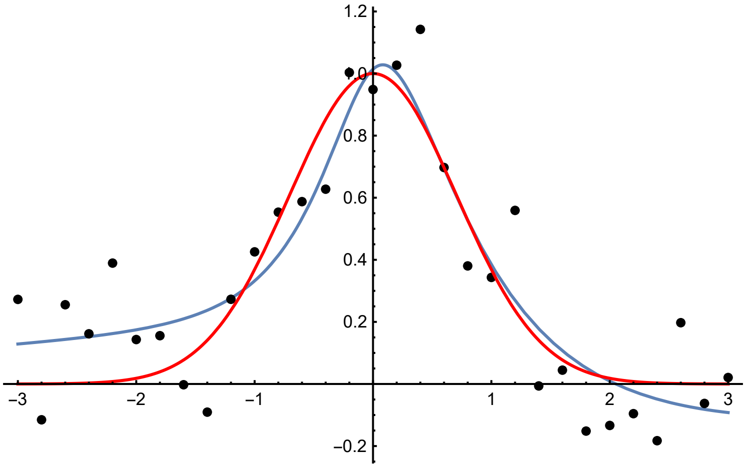
In[ ]:=
```
Show[Plot[net3[x], {x, -3, 3}, PlotRange → All], plot1]
```

Out[ ]=



In[ ]:=
```
Show[Plot[net3[x], {x, -3, 3}, PlotRange → All], plot1, gplot]
```

Out[ ]=



Option

Brownian Motion

Option Pricing

Option Pricing using Monte Carlo Method

Option Pricing using Neural Networks

In[∘]:=
```
Clear[S, K, r, T, σ, d1, d2]
```

In[∘]:=

$$d1 = \frac{\mathrm{Log}[S/K] + \left(r + \frac{1}{2}\sigma^2\right) T}{\sigma \sqrt{T}};$$

$$d2 = \frac{\mathrm{Log}[S/K] + \left(r - \frac{1}{2}\sigma^2\right) T}{\sigma \sqrt{T}};$$

$$\phi[u\_] := \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{u} e^{-v^2/2}\, dv$$

In[∘]:=

$$\mathrm{Call}[S\_, r\_, \sigma\_, K\_, T\_] = S * \phi[d1] - e^{-r*T} * K * \phi[d2]$$

Out[∘]=

$$\frac{1}{2} S \left(1 + \mathrm{Erf}\left[\frac{T\left(2 r + \sigma^2\right) + 2\,\mathrm{Log}\left[\frac{S}{K}\right]}{2\sqrt{2}\sqrt{T}\,\sigma}\right]\right) - \frac{1}{2} e^{-r T} K\, \mathrm{Erfc}\left[\frac{T\left(-2 r + \sigma^2\right) - 2\,\mathrm{Log}\left[\frac{S}{K}\right]}{2\sqrt{2}\sqrt{T}\,\sigma}\right]$$

In[∘]:=
```
Call[49, 0.05, 0.2, 50, 0.3846]
```

Out[∘]=
```
2.40046
```

In[∘]:=
```
callval = Call @@ {49, 0.05, 0.2, 50, 0.3846}
```

Out[∘]=
```
2.40046
```

Data generating for option values

```
In[ ]:= Clear[a1, a2, a3, a4, a5, dataall]
        dataall = Table[{a1, a2, a3, a4, a5} → Call @@ {a1, a2, a3, a4, a5}, {a1, 40, 60, 1},
            {a2, 0.01, 0.1, 0.01}, {a3, 0.1, 0.5, 0.1}, {a4, 40, 60, 1}, {a5, 0.1, 1, 0.1}];
        dataall = Flatten[dataall]
        Length[dataall]
```

```
Out[ ]=
  {{40, 0.01, 0.1, 40, 0.1} → 0.524595, {40, 0.01, 0.1, 40, 0.2} → 0.75355,
   ··· 220 496 ···, {60, 0.1, 0.5, 60, 0.9} → 13.536, {60, 0.1, 0.5, 60, 1.} → 14.356}
  Full expression not available (original memory size: 52.9 MB)
```

```
Out[ ]=
  220 500
```

Training data

```
In[ ]:= dataa2 = RandomSample[dataall];
```

NetChain[{$layer_1$, $layer_2$, …}]

specifies a neural net in which the output of $layer_i$ is connected to the input of $layer_{i+1}$.

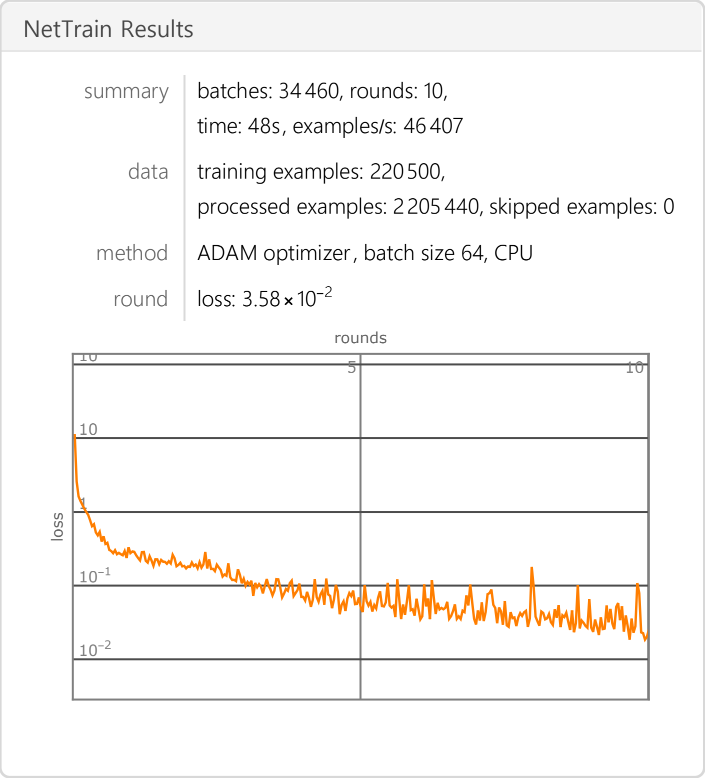
"ADAM" stochastic gradient descent using an adaptive learning rate that is invariant to diagonal rescaling of the gradients

"RMSProp" stochastic gradient descent using an adaptive learning rate derived from exponentially smoothed average of gradient magnitude

"SGD" ordinary stochastic gradient descent with momentum

```
In[◦]:= neta = NetChain[{150, Tanh, 150, Tanh, 100, Ramp, 1}, "Input" → 5, "Output" → "Scalar"];
        resulta1 = NetTrain[neta, dataa2, All, Method → "ADAM"]
Out[◦]=
```

NetTrainResultsObject[



NetTrain Results

summary | batches: 34 460, rounds: 10,
time: 48s, examples/s: 46 407

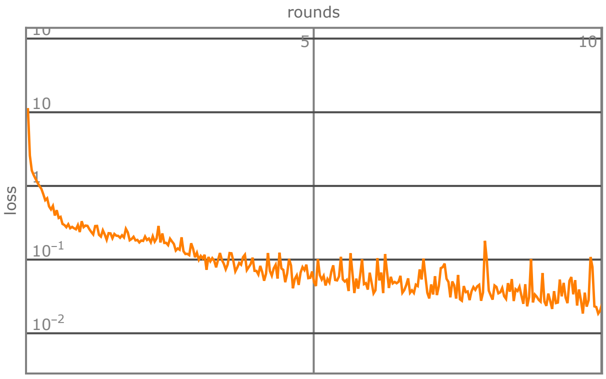
data | training examples: 220 500,
processed examples: 2 205 440, skipped examples: 0

method | ADAM optimizer, batch size 64, CPU

round | loss: 3.58×10^{-2}

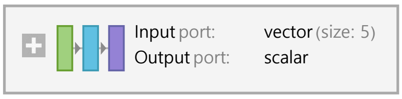
]

```
In[◦]:= resulta1["LossEvolutionPlot"]
Out[◦]=
```



```
In[◦]:= neta1 = resulta1["TrainedNet"]
Out[◦]=
```

NetChain[



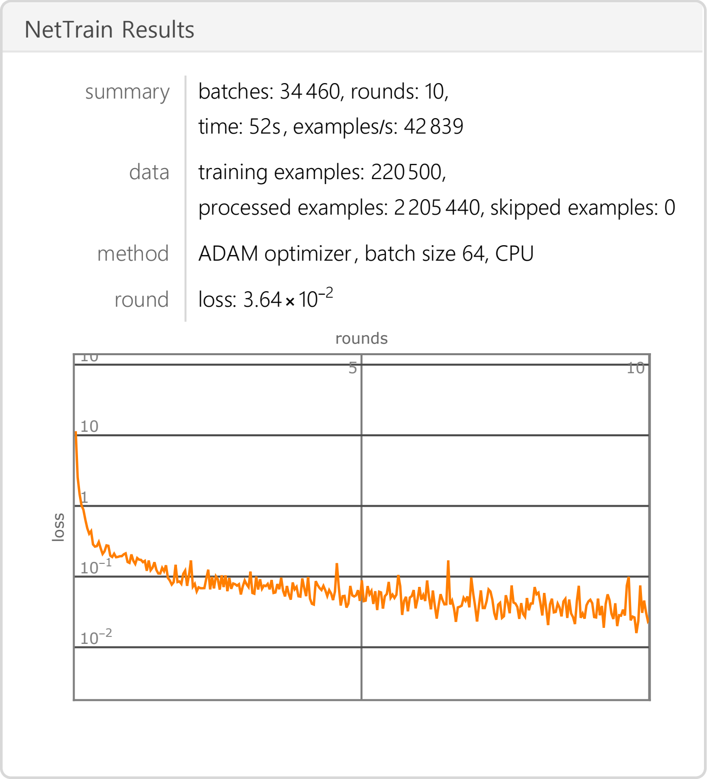
Input port: vector (size: 5)
Output port: scalar

]

```
In[◦]:= neta1[{49, 0.05, 0.2, 50, 0.3846}]
Out[◦]=
        2.39734
```

```
In[◦]:= Abs[% - callval] / (callval) * 100
Out[◦]=
        0.129849

            (* L2Regularization *)

In[◦]:= resultaL1 = NetTrain[neta, dataa2, All, Method → {"ADAM", "L2Regularization" → 0.001}]
        netaL1 = resultaL1["TrainedNet"]
        netaL1[{49, 0.05, 0.2, 50, 0.3846}]
        Abs[% - callval] / (callval) * 100
Out[◦]=
```

NetTrainResultsObject[



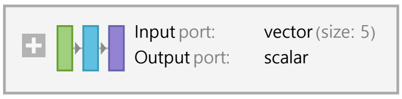
NetTrain Results

summary	batches: 34 460, rounds: 10, time: 52s, examples/s: 42 839
data	training examples: 220 500, processed examples: 2 205 440, skipped examples: 0
method	ADAM optimizer, batch size 64, CPU
round	loss: 3.64×10^{-2}

]

```
Out[◦]=
```

NetChain[



Input port:	vector (size: 5)
Output port:	scalar

]

```
Out[◦]=
        2.35049
Out[◦]=
        2.08182

In[◦]:= testX = Keys[dataa2];
        testY = Values[dataa2];
        meanTestLoss[net_] := SquaredEuclideanDistance[net[testX], testY] / Length[testX];

In[◦]:= meanTestLoss[neta1]
        meanTestLoss[netaL1]
Out[◦]=
        0.0162184
Out[◦]=
        0.0156701
```

In[∘]:=

```
neta = NetChain[{5, Tanh, 5, Tanh, 1}, "Input" → 5, "Output" → "Scalar"];
resultaa1 = NetTrain[neta, dataa2, All, Method → "ADAM"]
netaa1 = resultaa1["TrainedNet"]
netaa1[{49, 0.05, 0.2, 50, 0.3846}]
Abs[% - callval] / (callval) * 100
```

Training with test data

70% training

80% training

90% training

Comparison on the size of training data

```
(* Do[dataa2=RandomSample[dataa2],{n,100}]; *)
```

```
(* callval=Call@@{49,0.05,0.2,50,0.3846}; *)
```

In[∘]:=

```
testrate = Table[0.1 * n, {n, 1, 10, 0.5}]
```

Out[∘]=

```
{0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45,
 0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1.}
```

```
In[∘]:= j = 1; testrate = Table[0.1 * n, {n, 1, 10, 0.5}]; AA = {}; BB = {}; CC = {};
        While[j < Length[testrate],
          {traina2, testa2} = TakeDrop[dataa2, Length[dataall] * testrate⟦j⟧];
          nett = NetTrain[neta, traina2,
            ValidationSet → testa2, Method → "ADAM", MaxTrainingRounds → 10];
          AppendTo[AA, nett[{49, 0.05, 0.2, 50, 0.3846}]];
          AppendTo[BB, (Abs[nett[{49, 0.05, 0.2, 50, 0.3846}] - callval] / callval) * 100];
          AppendTo[CC, meanTestLoss[nett]];
          j++];
        AA
        BB
        CC
```

```
Out[∘]=
        {1.77885, 1.92936, 2.06591, 2.20978, 2.13107, 2.23176, 2.35828, 2.34089, 2.3894,
         2.31725, 2.48025, 2.33813, 2.2899, 2.32509, 2.34654, 2.3825, 2.37622, 2.37346}
```

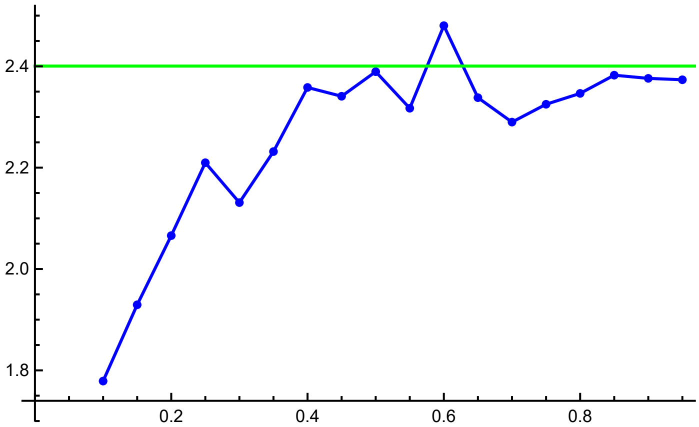
```
Out[∘]=
        {25.8957, 19.6254, 13.9367, 7.94363, 11.2226, 7.02791, 1.75714, 2.48159, 0.460661,
         3.46633, 3.32399, 2.59671, 4.60579, 3.1398, 2.24615, 0.748416, 1.01001, 1.12476}
```

```
Out[∘]=
        {0.264887, 0.180531, 0.138248, 0.10225, 0.122876, 0.0569396,
         0.0598894, 0.0429713, 0.0350557, 0.0240843, 0.0459525, 0.0414602,
         0.0222191, 0.0244577, 0.0213035, 0.0186292, 0.0145992, 0.0172346}
```

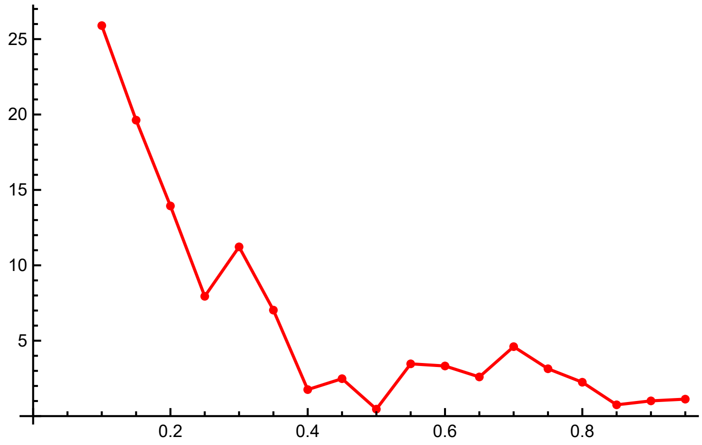
```
In[∘]:= xx = TakeDrop[testrate, -1]⟦2⟧;
        AAL = Thread[{xx, AA}];
        BBL = Thread[{xx, BB}];
        CCL = Thread[{xx, CC}];
```

```
In[∘]:= Show[{ListPlot[AAL, PlotStyle → Blue, Joined → True, Mesh → All],
          Plot[callval, {x, 0, 18}, PlotStyle → Green]}]
```

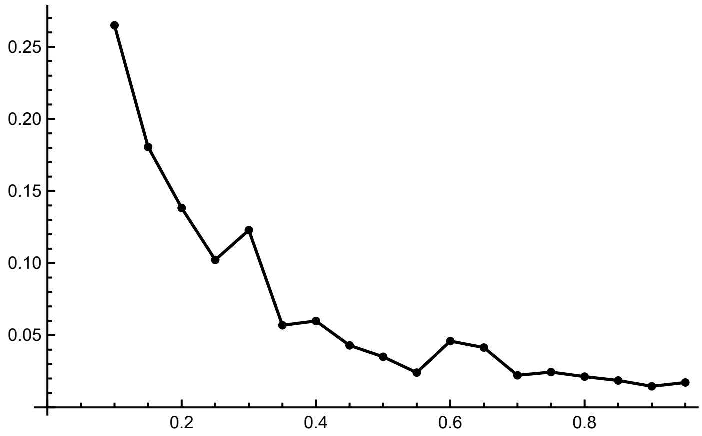
Out[∘]=



```
In[•]:= ListPlot[BBL, PlotStyle → Red, Joined → True, Mesh → All]
Out[•]=
```



```
In[•]:= ListPlot[CCL, PlotStyle → Black, Joined → True, Mesh → All]
Out[•]=
```



```
In[•]:= Length[testrate]
Out[•]=
19
```

Comparison on the size of training data without max round

```
(* Do[dataa2=RandomSample[dataa2],{n,100}]; *)

(* callval=Call@@{49,0.05,0.2,50,0.3846}; *)
```

```
In[∘]:= j = 1;
        testrate = Table[0.1 * n, {n, 1, 10, 0.5}];
        AAw = {};
        BBw = {};
        CCw = {};
        While[j < Length[testrate],
          {traina2, testa2} = TakeDrop[dataa2, Length[dataall] * testrate⟦j⟧];
          nett = NetTrain[neta, traina2, ValidationSet → testa2, Method → "ADAM"];
          AppendTo[AAw, nett[{49, 0.05, 0.2, 50, 0.3846}]];
          AppendTo[BBw, (Abs[nett[{49, 0.05, 0.2, 50, 0.3846}] - callval] / callval) * 100];
          AppendTo[CCw, meanTestLoss[nett]];
          j++];
        AAw
        BBw
        CCw
```

```
Out[∘]=
        {2.37205, 2.42548, 2.47838, 2.35137, 2.32867, 2.27319, 2.28488, 2.43241, 2.3894,
         2.31725, 2.48025, 2.33813, 2.2899, 2.32509, 2.34654, 2.3825, 2.37622, 2.37346}
```

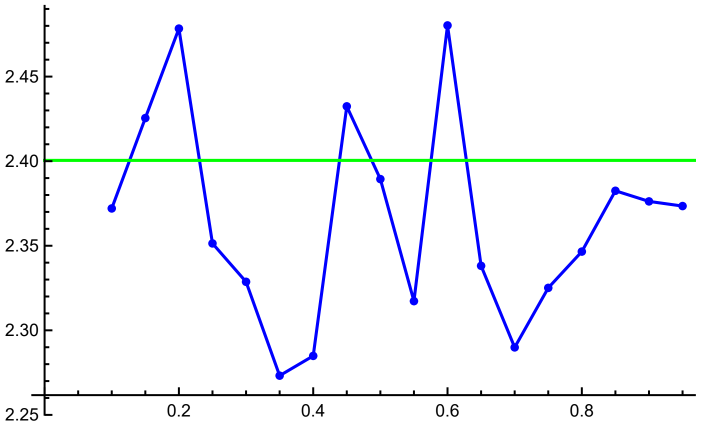
```
Out[∘]=
        {1.18349, 1.04227, 3.24606, 2.04488, 2.99059, 5.3019, 4.8149, 1.33095, 0.460661,
         3.46633, 3.32399, 2.59671, 4.60579, 3.1398, 2.24615, 0.748416, 1.01001, 1.12476}
```

```
Out[∘]=
        {0.0244919, 0.0389958, 0.0234126, 0.0257422, 0.0628817, 0.0258112,
         0.0472577, 0.0243519, 0.0350557, 0.0240843, 0.0459525, 0.0414602,
         0.0222191, 0.0244577, 0.0213035, 0.0186292, 0.0145992, 0.0172346}
```

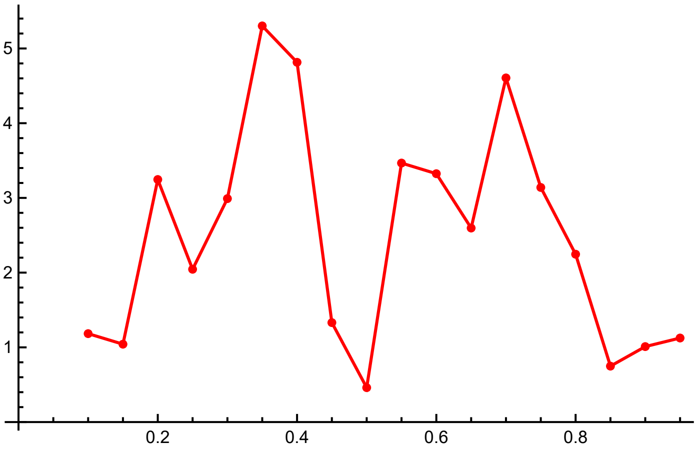
```
In[∘]:= AALw = Thread[{xx, AAw}];
        BBLw = Thread[{xx, BBw}];
        CCLw = Thread[{xx, CCw}];
```

```
In[∘]:= Show[{ListPlot[AALw, PlotStyle → Blue, Joined → True, Mesh → All],
          Plot[callval, {x, 0, 18}, PlotStyle → Green]}]
```

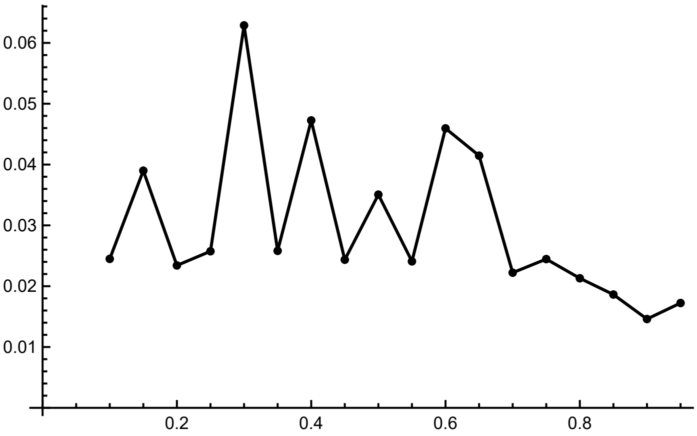
Out[∘]=



```
In[ ]:= ListPlot[BBLw, PlotStyle → Red, Joined → True, Mesh → All]
Out[ ]=
```



```
In[ ]:= ListPlot[CCLw, PlotStyle → Black, Joined → True, Mesh → All]
```



```
{49, 0.05, 0.2, 50, 0.3846}
```

```
In[ ]:= neta29[{49, 0.05, 0.2, 50, 0.38}]
Out[ ]=
        2.35689
```

Training data using GPU

Training with test data using GPU

Comparison on the size of training data using GPU

```
In[∘]:= j = 1; testrate = Table[0.1 * n, {n, 1, 10, 0.5}]; AAG = {}; BBG = {};
        CCG = {};
        While[j < Length[testrate],
          {traina2, testa2} = TakeDrop[dataa2, Length[dataall] * testrate⟦j⟧];
          nett = NetTrain[neta, traina2, ValidationSet → testa2,
            Method → "ADAM", MaxTrainingRounds → 10, TargetDevice → "GPU"];
          AppendTo[AAG, nett[{49, 0.05, 0.2, 50, 0.3846}]];
          AppendTo[BBG, (Abs[nett[{49, 0.05, 0.2, 50, 0.3846}] - callval] / callval) * 100];
          AppendTo[CCG, meanTestLoss[nett]];
          j++];
        AAG
        BBG
        CCG
```

```
Out[∘]=
        {1.87647, 1.99027, 1.93657, 2.21983, 2.11092, 2.20737, 2.18173, 2.44253, 2.3533,
         2.23777, 2.32034, 2.31561, 2.34591, 2.47656, 2.46827, 2.41508, 2.46416, 2.32508}
```

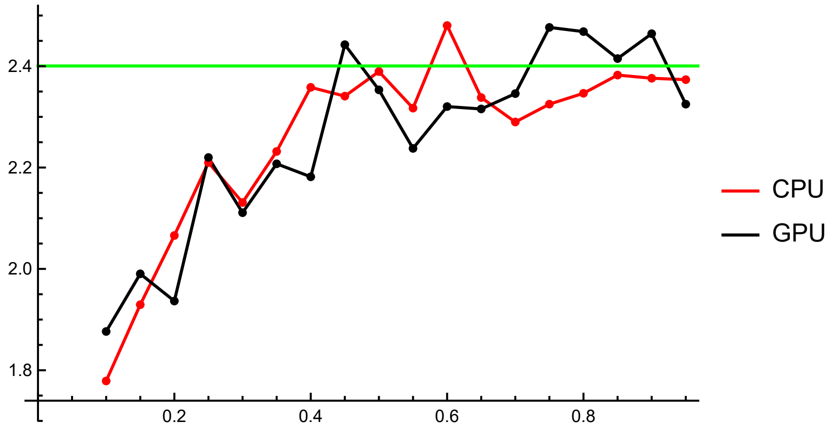
```
Out[∘]=
        {21.8287, 17.0882, 19.3252, 7.52491, 12.0619, 8.04379, 9.11192, 1.75264, 1.9647,
         6.77765, 3.3379, 3.53475, 2.27234, 3.16999, 2.82469, 0.608978, 2.65344, 3.14027}
```

```
Out[∘]=
        {0.249937, 0.179354, 0.144602, 0.102996, 0.125443, 0.0610667,
         0.0550897, 0.0456403, 0.0337748, 0.0307407, 0.03595, 0.0453212,
         0.0215164, 0.0458719, 0.0283106, 0.0171917, 0.0191037, 0.015763}
```

```
In[∘]:= AAGL = Thread[{xx, AAG}];
        BBGL = Thread[{xx, BBG}];
        CCGL = Thread[{xx, CCG}];
```

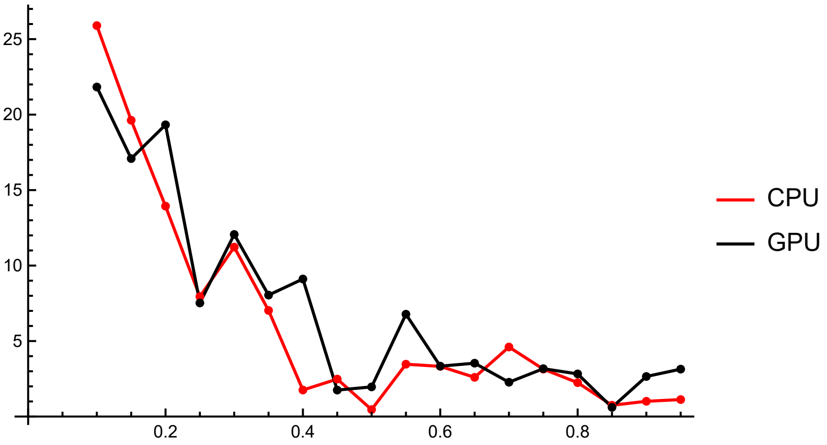
```
In[∘]:= Show[{ListPlot[{AAL, AAGL}, PlotStyle → {Red, Black}, Joined → True, Mesh → All,
           PlotLegends → {"CPU", "GPU"}], Plot[callval, {x, 0, 20}, PlotStyle → Green]}]
```

Out[∘]=



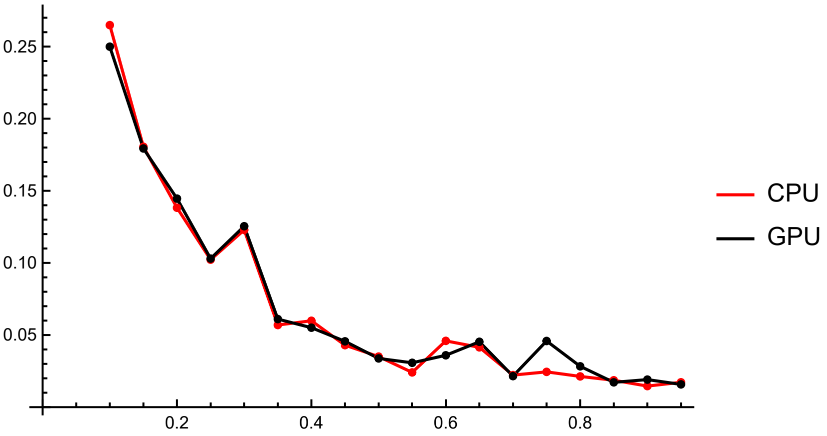
In[ ]:=
```
ListPlot[{BBL, BBGL}, PlotStyle → {Red, Black},
  Joined → True, Mesh → All, PlotLegends → {"CPU", "GPU"}]
```

Out[ ]=



In[ ]:=
```
ListPlot[{CCL, CCGL}, PlotStyle → {Red, Black},
  Joined → True, Mesh → All, PlotLegends → {"CPU", "GPU"}]
```

Out[ ]=



Comparison on the size of training data using GPU without Max-Rd

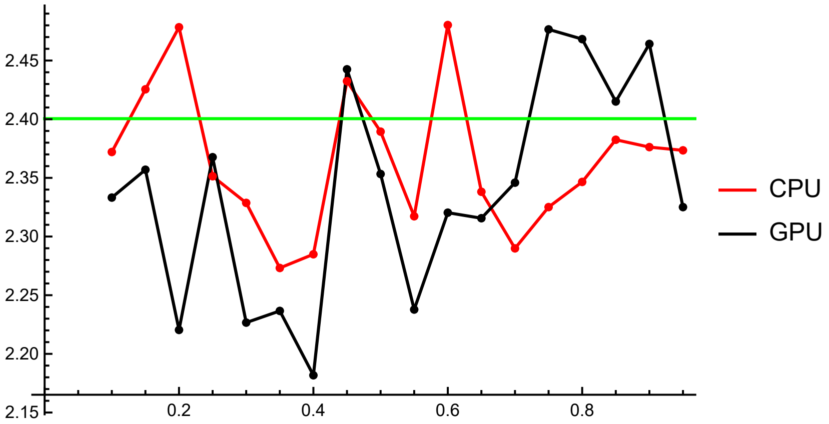
```
In[∘]:= j = 1; testrate = Table[0.1 * n, {n, 1, 10, 0.5}]; AAGw = {}; BBGw = {};
        CCGw = {};
        While[j < Length[testrate],
          {traina2, testa2} = TakeDrop[dataa2, Length[dataall] * testrate⟦j⟧];
          nett = NetTrain[neta, traina2,
            ValidationSet → testa2, Method → "ADAM", TargetDevice → "GPU"];
          AppendTo[AAGw, nett[{49, 0.05, 0.2, 50, 0.3846}]];
          AppendTo[BBGw, (Abs[nett[{49, 0.05, 0.2, 50, 0.3846}] - callval] / callval) * 100];
          AppendTo[CCGw, meanTestLoss[nett]];
          j++];
        AAGw
        BBGw
        CCGw
```

```
Out[∘]=
        {2.33319, 2.35696, 2.22039, 2.36762, 2.22664, 2.2366, 2.18173, 2.44253, 2.3533,
         2.23777, 2.32034, 2.31561, 2.34591, 2.47656, 2.46827, 2.41508, 2.46416, 2.32508}

Out[∘]=
        {2.80227, 1.8124, 7.50157, 1.36798, 7.24133, 6.82631, 9.11192, 1.75264, 1.9647,
         6.77765, 3.3379, 3.53475, 2.27234, 3.16999, 2.82469, 0.608978, 2.65344, 3.14027}

Out[∘]=
        {0.0337538, 0.0423014, 0.0530205, 0.0411814, 0.0674355, 0.056641,
         0.0550897, 0.0456403, 0.0337748, 0.0307407, 0.03595, 0.0453212,
         0.0215164, 0.0458719, 0.0283106, 0.0171917, 0.0191037, 0.015763}
```

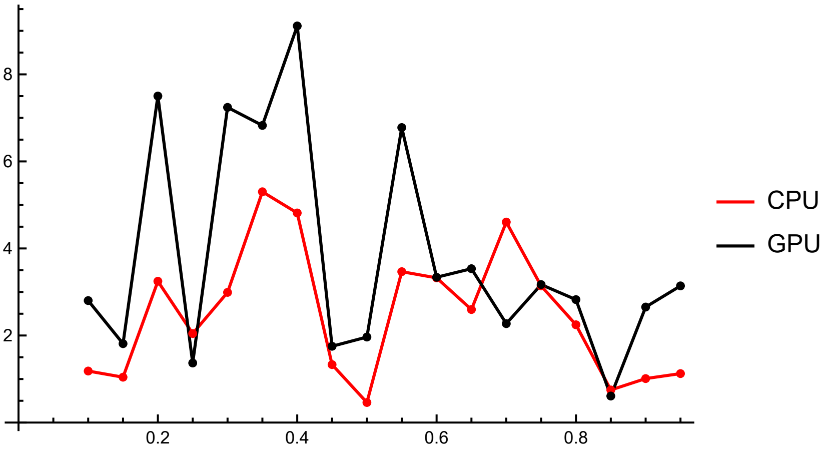
```
In[∘]:= AAGLw = Thread[{xx, AAGw}];
        BBGLw = Thread[{xx, BBGw}];
        CCGLw = Thread[{xx, CCGw}];
```

```
In[∘]:= Show[{ListPlot[{AALw, AAGLw}, PlotStyle → {Red, Black}, Joined → True, Mesh → All,
           PlotLegends → {"CPU", "GPU"}], Plot[callval, {x, 0, 20}, PlotStyle → Green]}]
```

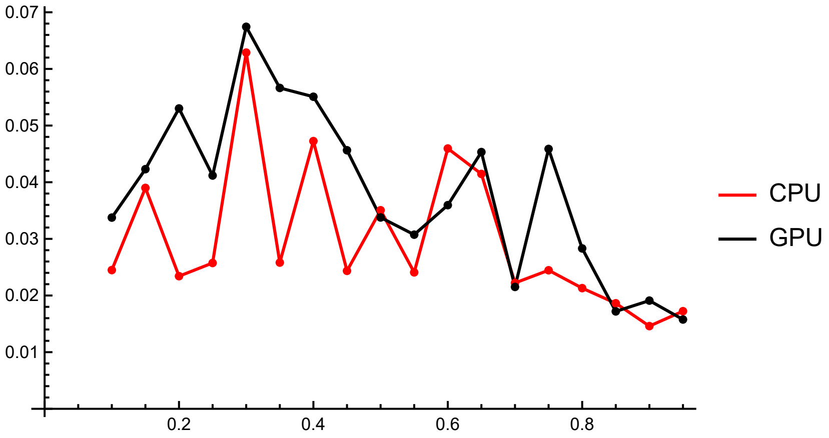
Out[∘]=



```
In[∘]:= ListPlot[{BBLw, BBGLw}, PlotStyle → {Red, Black},
          Joined → True, Mesh → All, PlotLegends → {"CPU", "GPU"}]
Out[∘]=
```



```
In[∘]:= ListPlot[{CCLw, CCGLw}, PlotStyle → {Red, Black},
          Joined → True, Mesh → All, PlotLegends → {"CPU", "GPU"}]
Out[∘]=
```


CPU
GPU